System design: Web Crawler

Suresh Podeti
5 min read · Oct 25, 2023

Introduction

A web crawler, an Internet bot, systematically searches the web for content, starting with seed URLs. This process, known as crawling, stores the acquired content for subsequent use. Search engines use this stored data for indexing and ranking.

This specific discussion focuses on web crawling and does not cover indexing and ranking stages in search engines.

Requirements

Functional

- **Crawling —** The system should scour the WWW, spanning from a list of seed URLs provided initially by the system administrator.
- **Storing —** The system should be able to extract and store the content of a URL in a **blob store**. This makes that URL and its content processable by the search engines for indexing and ranking purposes.
- **Scheduling:** Since crawling is a process that's repeated, the **system should have regular scheduling** to update its blob stores' records

Non-functional

- **Scalability —** The system should inherently be distributed and multithreaded, because it has to fetch hundreds of millions of web documents.
- **Consistency**: Since our system involves multiple crawling workers, having data consistency among all of them is necessary.

Traversal time

Since the traversal time is just as important as the storage requirements, let's calculate the approximate time for one-time crawling. Assuming that the average HTTP traversal per webpage is **60 ms**, the time to traverse all **5 billion pages** will be:

Total traversal time=5 *Billion*×60 *ms*=0.3 *Billion seconds*= **9.5 *years***

It'll take approximately 9.5 years to traverse the whole Internet while using **one instance of crawling**, but we want to achieve our goal in one day. We can accomplish this by designing our system to support **multi-worker architecture** and divide the tasks among multiple workers running on different servers.

Components

Scheduler

This is one of the key building blocks that **schedules URLs** for crawling. It's composed of two units: a priority queue and a relational database.

1. **A priority queue (URL frontier)**: The queue hosts **URLs** that are made **ready for crawling** based on the two properties associated with each entry: priority and updates frequency.
2. **Relational database**: It **stores all the URLs** along with the two associated parameters mentioned above. The database gets populated by new requests from the following two input streams:

- The user's *added URLs*, which include seed and runtime added URLs.
- The crawler's *extracted URLs*.

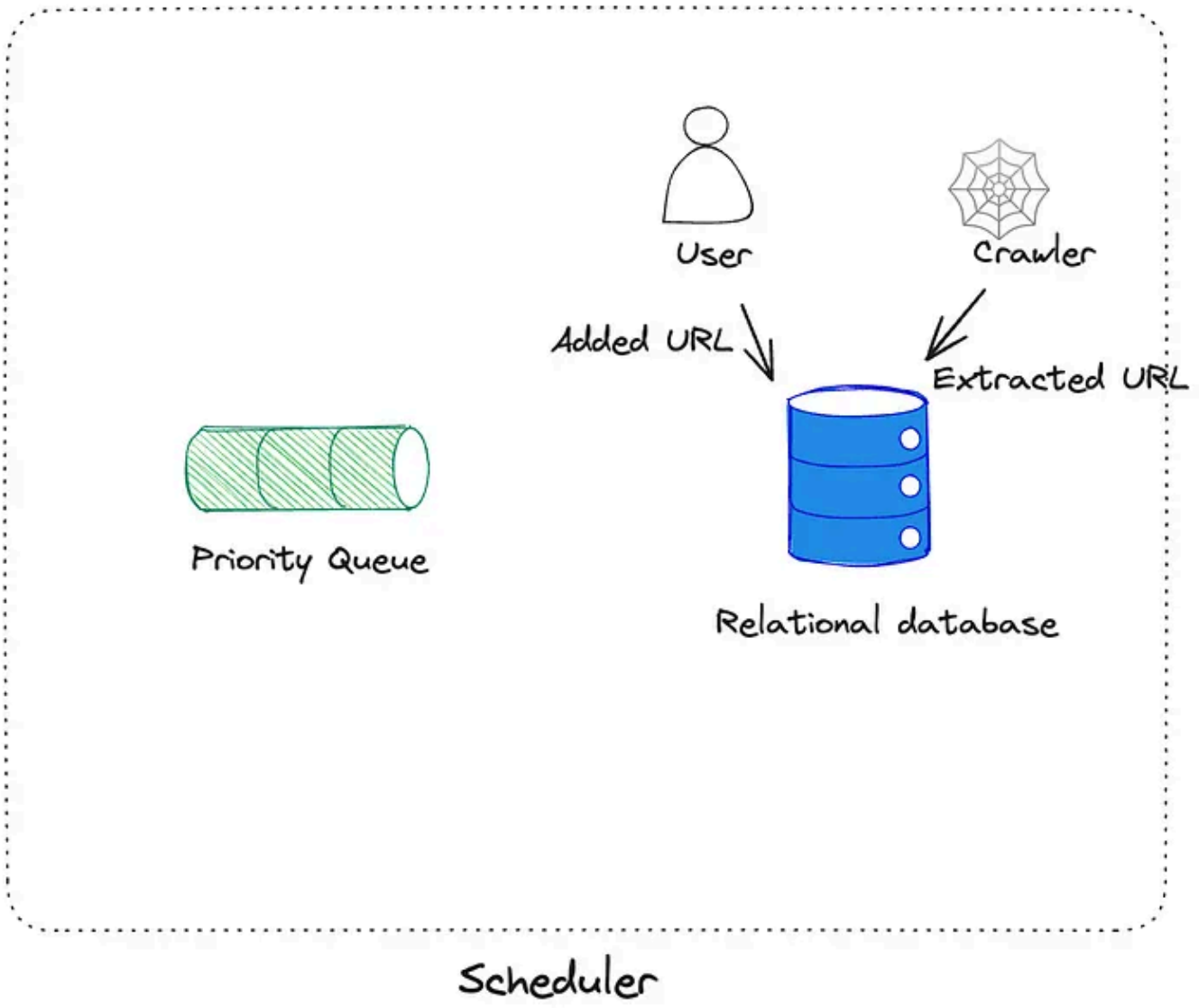

Fig 1.0: Scheduler component

DNS resolver

The web crawler needs a DNS resolver to **map hostnames to IP addresses** for HTML content fetching. Since **DNS lookup is a time-consuming** process, a **better approach** is to **create a customized DNS resolver** and cache frequently-used IP addresses within their time-to-live because they're bound to change after their time-to-live.

HTML fetcher

The HTML fetcher initiates communication with the server that's hosting the URL(s). It **downloads the file content** based on the underlying communication protocol. We focus mainly on the *HTTP protocol* for textual content, but the *HTML fetcher* is easily extendable to other communication protocols.

Service host

This component acts as the brain of the crawler and is composed of worker instances. There are three main tasks that this service host/crawler performs:

- It handles the **multi-worker architecture** of the crawling operation. Based on the availability, **each worker** communicates with the URL frontier to **dequeue the next available URL for crawling**.
- Each worker is **responsible for acquiring the DNS resolutions** of the incoming URLs from the DNS resolver.
- Each worker acts as a gateway between the scheduler and the HTML fetcher by **sending** the necessary **DNS resolution information** to the **HTML fetcher** for communication initiation.

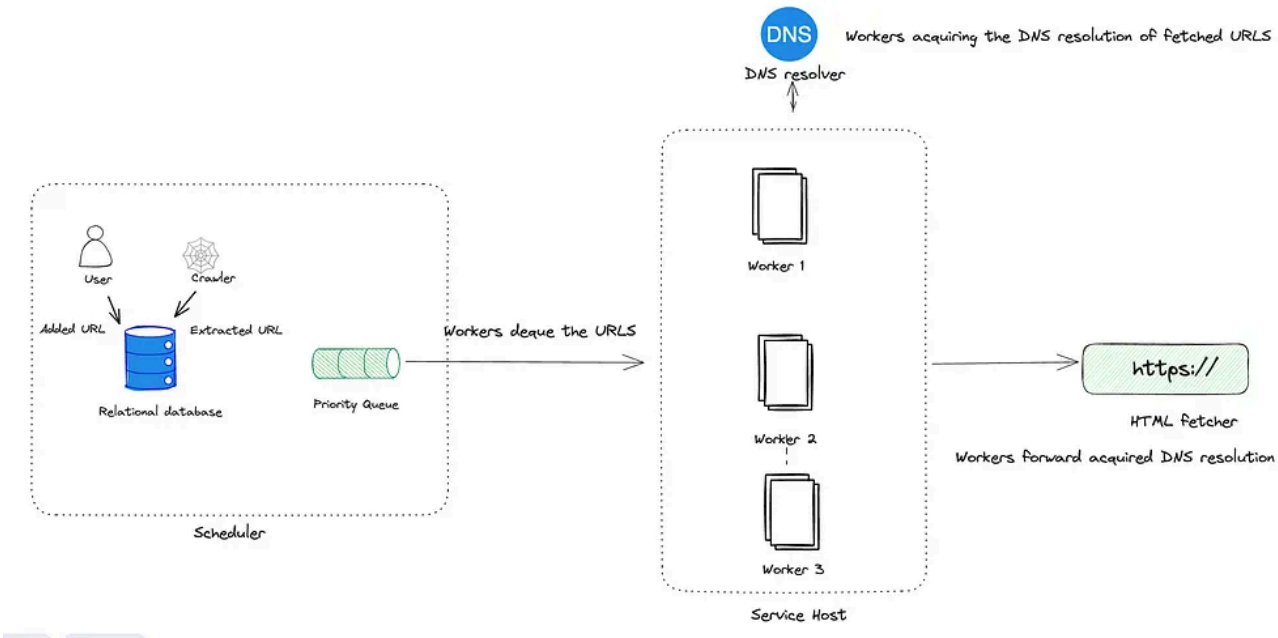

Fig 2.0: Service host integration with other components

Extractor

Once the HTML fetcher gets the web page, the next step is to extract two things from the webpage: **URLs and the content**. The extractor sends the extracted URLs directly and the content with the **document input stream (DIS)** to the duplicate eliminator. **DIS is a cache** that's used to store the extracted document, so that other components can access and process it. Over here, we can use **Redis** as our cache choice because of its advanced data structure functionality.

Once it's verified that the duplicates are absent in the data stores, the extractor sends the **URLs to the task scheduler** that contains the URL frontier and stores the content in **blob storage** for indexing purposes

Duplicate eliminator

The web's interconnected nature leads to the possibility of duplicate content from different URLs. To prevent resource wastage, a deduplication component checks for duplicates. It calculates checksums for extracted URLs and compares them to stored checksums. Matches are discarded, while new entries are added to the database.

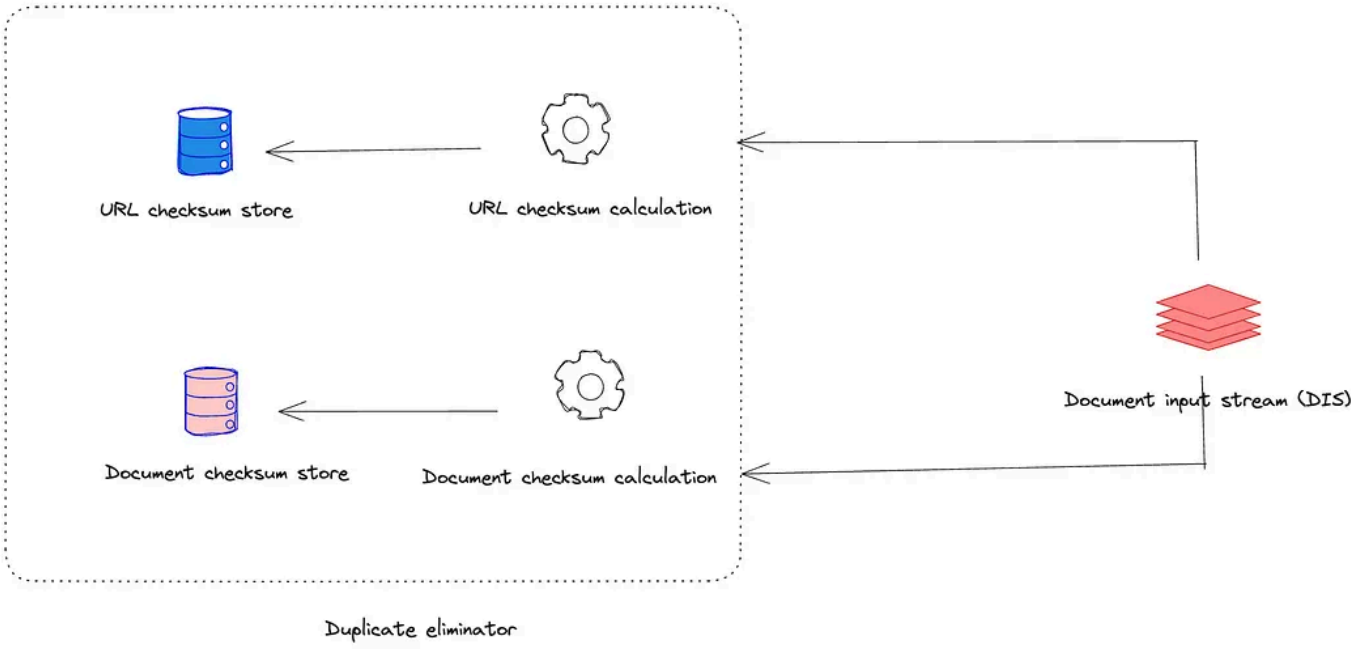

Fig 3.0: Duplicate eliminator design

The duplicate eliminator **repeats the same process with the extracted content** and adds the new webpage's checksum value in the document checksum data store for future matchings.

Blob store

Since a web crawler is the **backbone** of a **search engine, storing and indexing** the fetched content and relevant metadata is immensely important. The design needs to have a distributed storage, such as a **blob store**, because we need to store large volumes of **unstructured data**.

Final design

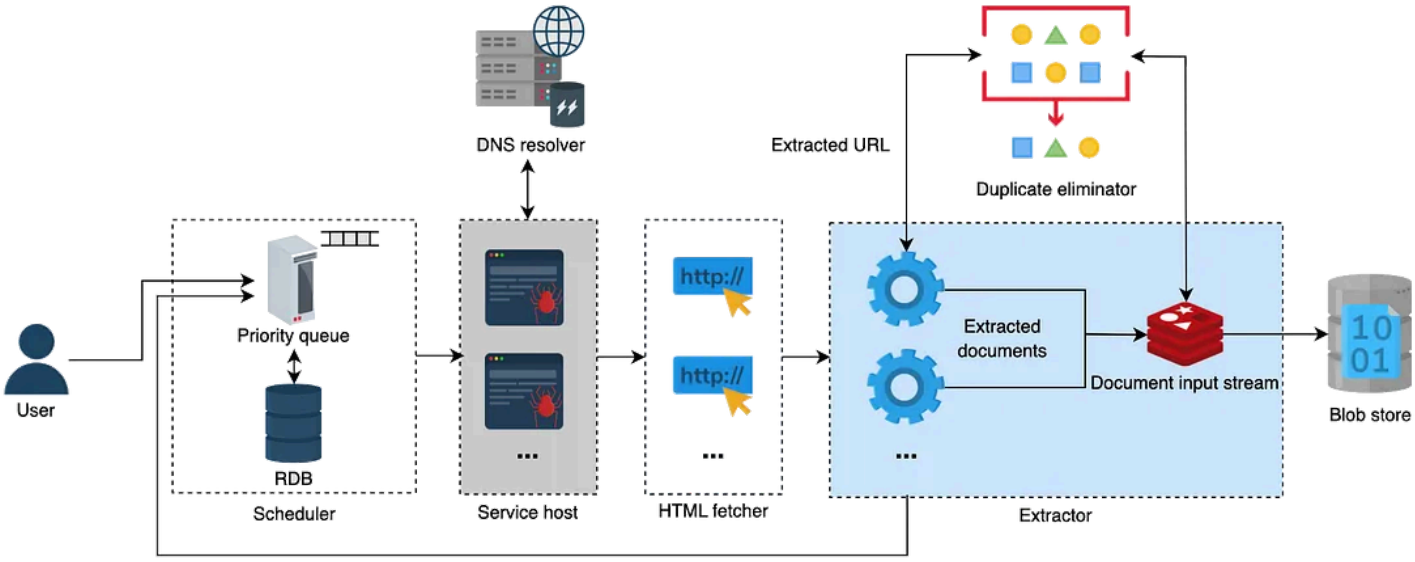

Fig 4.0: Final design of the web crawler

Evaluation

Scalability

- The system is scalable to handle the ever-increasing number of URLs. It includes all the required resources, including schedulers, web crawler workers, HTML fetchers, extractors, and blob stores, which are added/removed on demand.
- In the case of a distributed URL frontier, the system utilizes consistent hashing to distribute the hostnames among various crawling workers, where each worker is running on a server. With this, adding or removing a crawler server isn't a problem.

Consistency

Our system consists of several crawling workers. Data consistency among crawled content is crucial. So, to avoid data inconsistency and crawl duplication, our system computes the checksums of URLs and documents and compares them with the existing checksums of the URLs and documents in the *URL* and *document checksum* data stores, respectively.

Apart from deduplication, to ensure the data consistency by fault-tolerance conditions, all the servers can checkpoint their states to a backup service, such as Amazon S3 or an offline disk, regularly.

Web Crawler | System Design Interview | Software Architect | Software Architecture | Distributed Systems

Edit profile

Written by Suresh Podeti

1.2K Followers